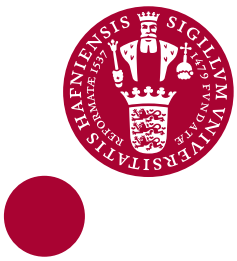

Sampling Random Bioinformatics Puzzles using Adaptive Probability Distributions

Have, Christian Theil; Appel, Emil Vincent; Bork-Jensen, Jette; Lassen, Ole Torp

Published in:
CEUR Workshop Proceedings

Publication date:
2016

Document version
Publisher's PDF, also known as Version of record

Sampling Random Bioinformatics Puzzles using Adaptive Probability Distributions

Christian Theil Have[1], Emil Vincent Appel[1], Jette Bork-Jensen[1], and Ole Torp Lassen[2]

[1] Novo Nordisk Foundation Center for Basic Metabolic Research, Section of Metabolic Genetics, University of Copenhagen, Denmark.
[2] Roskilde University, Roskilde, Denmark

Abstract. We present a probabilistic logic program to generate an educational puzzle that introduces the basic principles of next generation sequencing, gene finding and the translation of genes to proteins following the central dogma in biology. In the puzzle, a secret "protein word" must be found by assembling DNA from fragments (reads), locating a gene in this sequence and translating the gene to a protein.
Sampling using this program generates random instance of the puzzle, but it is possible constrain the difficulty and to customize the secret protein word. Because of these constraints and the randomness of the generation process, sampling may fail to generate a satisfactory puzzle. To avoid failure we employ a strategy using adaptive probabilities which change in response to previous steps of generative process, thus minimizing the risk of failure.

Keywords: PRISM, bioinformatics, sampling

1 Introduction

Assemble-yourself[3] is a paper-printable "bioinformatics" puzzle which is intended to be fun as well as educational – it introduces concepts in genetics and Next Generation Sequencing (NGS) [1]. We have successfully used *assemble-yourself* in a workshop setting, where groups of participants competed against each other. They were quite engaged and reported to have a lot of fun while also learning. The idea of the game is to assemble a DNA sequence from NGS reads and translate it into amino acids to find a secret word. The entire game is printed on paper. This includes the reads (cut out paperstrips) and a board which serves scaffold to assemble the reads and writing down the sequence. Once the reads have been assembled and a consensus sequence is found, the consensus sequence is translated to amino acid sequences in both DNA strands. The letters of the sequence contains a secret word embedded within an open reading frame.

The game is generated by a probabilistic logic program and many aspects of the game can be configured prior to generation. For instance, the secret protein

[3] https://github.com/cth/assemble-yourself

word can be changed and the minimal read depth — the number of reads covering any position — can be configured. In order to ensure satisfaction of constraints in a computationally efficient way, we introduce a heuristic scheme to adapt random switch probabilities based previous states of the program.

2 Description of the game

In the style of logic programming, we begin with the goal. The goal of the game is to find a particular protein word. For a given game instance, a mutated version of the specified protein word — where some letters (amino acids) are randomly substituted — is generated and reported in the background story, c.f., figure 1.

Recently an interesting protein with the amino acid sequence *ILP* was found in the bacteria *S. Equencia*. It is now to be determined if a homologue exists in the species *B. Ionformatica*.
To determine this a lab amplificied a relevant part of the DNA of *B. Ionformatica* using PCR primers flanking the gene in *S. Equencia* which are believed to be highly conserved also in *B. Ionformatica*, although the sequence of *B. Ionformatica* is currently not known. The amplified DNA was sequenced using Ullamini LoSeq next generation sequencing tech. The quality of the reads are not perfect – read errors resulting in random "mutations" are expected in one out of twenty bases.
As a bioinformatician you are given the task to find out if *B. Ionformatica* has a homologue of the protein *ILP* and determine how its amino acid sequence differs in *B. Ionformatica*. However, the high performance moon grid engine supercluster is currently down (as it sometimes is) and you have to do it all by hand. Fortunately, you have printed all the reads. You task is as follows: 1) Perform de-novo assembly of all the reads, 2) Find open reading frames that may contain a gene, 3) Find the amino acid sequence of any such gene to determine if it could be a homologue to *ILP*, 4) Report your finding and claim eternal fame.

Fig. 1. The background story

The amino acids are reflected in a corresponding the DNA sequence of nucleotide bases forming a gene, where each triplet of DNA bases (called a codon) correspond to an amino acid. Only part of the DNA sequence is used to encode the protein word. It is flanked by random DNA on both sides, but the offset of the encoded protein is randomly determined. A given amino acid may be represented by one of multiple coden triplets, a particular codon may indicate the start of the gene and another is used to indicate the end of it (see figure 2, and e.g., https://en.wikipedia.org/wiki/Genetic_code for details). The DNA sequence can be read either left-to-right or right-to-left, but in the latter case the DNA sequence should be complimented, i.e., the bases should be replaced by the one they pair with in the double-helical DNA molecule ($A \leftrightarrow T$ and $G \leftrightarrow C$). Yet another complication is that codon triplets may start in any position.

Part of the puzzle is to find possible genes in DNA, translate the nucleotide bases into an amino sequence to find the secret protein word. Once found the game is completed. However, before doing this, the player is faced with solving the problem of assembling the sequencing from a number of overlapping subsequences of the sequence (*reads*). This mimics the situation that arise when using NGS technologies, where the assembly process is handled by computationally intensive algorithms. The user however, is tasked with doing this by hand for a small DNA sequence. A scaled down instance of the game is shown in figure 3.

Second base in codon

First base in codon	T	C	A	G	Third base in codon
T	TTT Phe F	TCT Ser S	TAT Tyr Y	TGT Cys Y	T
	TTC Phe F	TCC Ser S	TAC Tyr Y	TGC Cys Y	C
	TTA Leu L	TCA Ser S	TAA Stop *	TGA Stop *	A
	TTG Leu L	TCG Ser S	TAG Stop *	TGG Trp W	G
C	CTT Leu L	CCT Pro P	CAT His H	CGT Arg R	T
	CTC Leu L	CCC Pro P	CAC His H	CGC Arg R	C
	CTA Leu L	CCA Pro P	CAA Gln Q	CGA Arg R	A
	CTG Leu L	CCG Pro P	CAG Gln Q	CGG Arg R	G
A	ATT Ile I	ACT Thr T	AAT Asn N	AGT Ser S	T
	ATC Ile I	ACC Thr T	AAC Asn N	AGC Ser S	C
	ATA Ile I	ACA Thr T	AAA Lys K	AGA Arg R	A
	ATG Met M	ACG Thr T	AAG Lys K	AGG Arg R	G
G	GTT Val V	GCT Ala A	GAT Asp D	GGT Gly G	T
	GTC Val V	GCC Ala A	GAC Asp D	GGC Gly G	C
	GTA Val V	GCA Ala A	GAA Glu E	GGA Gly G	A
	GTG Val V	GCG Ala A	GAG Glu E	GGG Gly G	G

Fig. 2. Standard genetic code. The table shows how triplets of nucleic acid bases correspond to different amino acids. Besides the codon ATG which always codes for methionine, alternatively TTG, CTG, ATT, ATC, ATA and GTG can serve as initiation codons, in which case they are translated as methionine rather than the amino acid indicated. In this puzzle we do require a leading methionine but do not consider it as part of the solution word.

3 Description of the program

We have written the program to generate puzzles in the PRISM language [2] — a probabilistic dialect of Prolog — although none of the advanced inference procedures of PRISM are used. The program executes in *sampling* mode to generate a random puzzle. Unlike the execution of a usual Prolog program, select non-deterministic choice points are replaced with committed random choices beyond which backtracking do not occur. Since the sampling execution commits to these random choices, unification failures cannot under normal circumstances be undone by backtracking, and the execution may fail if specified constraints are not satisfied. The output of the program is a LaTeXdocument which is compiled to printable PDF document which constitutes the puzzle. The generation of the LaTeXdocument is not probabilistic, but is implemented as a usual Prolog program which takes as input the probabilistic choices made in the sampling part of program.

The part of the puzzle which involves the protein word corresponding amino acid sequence and its underlying DNA sequence can be generated by a series of probabilistic choices on where to place the protein word, on generation of the

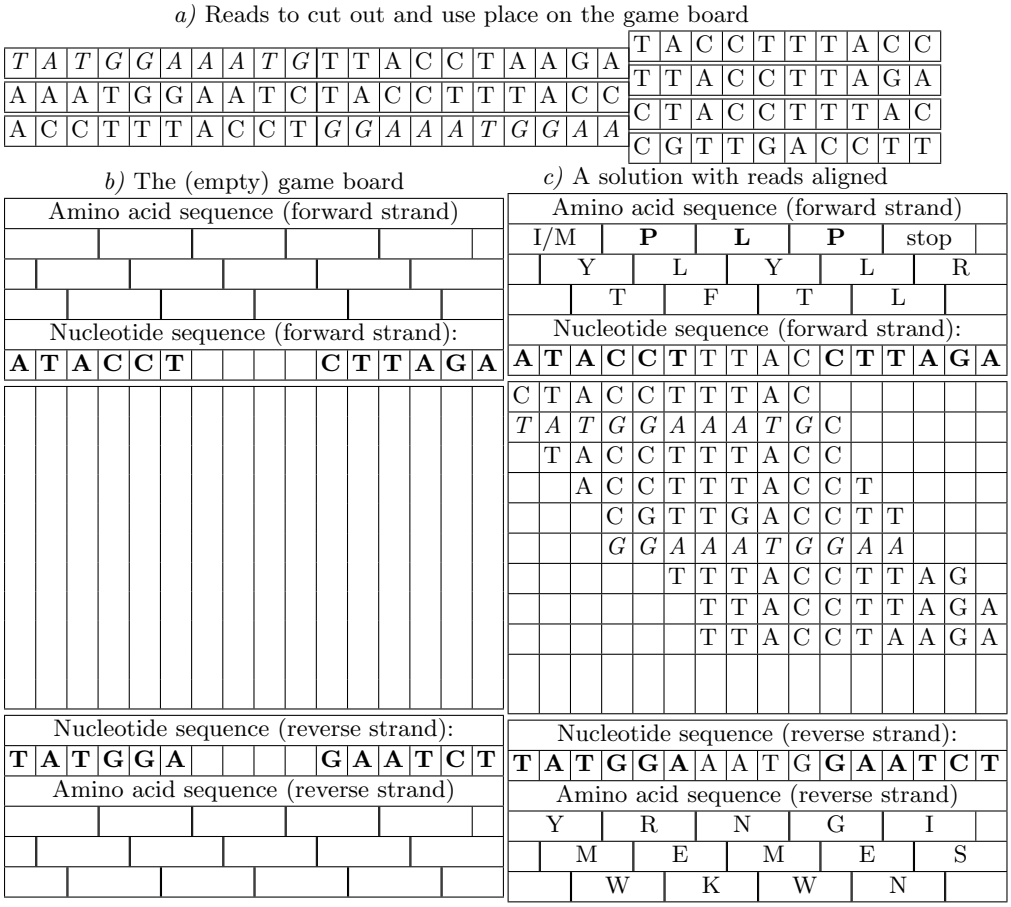

Fig. 3. A scaled down version of puzzle game along with a solution to the read assembly problem. The reads *(a)* should be placed on the board *(b)* to form a solution similar to *(c)*. The secret word (**PLP**) is marked in bold. The amino acids have been added by using the codon table in figure 2.

protein word with probability of point mutations as well as corresponding and additional point mutations in the underlying DNA sequence.

We must, however, control the number of mutations that are introduced in the protein word. If no mutations occur, the word will be the same as the the homologue protein in the background story, whereas if too many mutations occur, it may be difficult to determine if a discovered protein is correct. Hence, we constrain the number of mutations in the protein word to a specified range. To address this constraint we use PRISMs `soft_msw` construct [3], which provide a backtrack-able random switch mechanism in contrast to the usual committed choice construct for creating random switches, `msw`. Upon backtracking, the previously selected outcome of the switch is (temporarily) removed and probability

distribution over remaining outcomes is re-normalized before an alternate choice is attempted.

Note that it is not strictly necessary to rely on backtracking to solve this particular problem. An alternative failure-free approach would be to randomly choose the number of mutations from the specified range, randomly determine at which positions these mutations should occur and then introduce these mutations in the protein word.

In the generation of reads another global constraint come into play – each position of the DNA sequence must be covered by a specified minimum number of reads – the depth. At the same time, the total number of reads is constrained and generally kept to a minimum in order to balance the level fun and difficulty in the puzzle.

Reads are generated by a recursive predicate in which the termination case of the predicate specifies the condition that all positions must have the required minimum depth and the recursive case generates a random read, aligns it to the DNA sequence and updates a *depth vector*, $d_1 \ldots d_n$, for each position $1 \ldots n$ in the DNA sequence. It is not possible for this predicate to fail, but it may produce too many reads causing an assertion to fail in the calling predicate.

Initially we used `soft_msw` in our implementation, which would backtrack upon failure, but this approach exhibited poor performance for larger board sizes with a low cap on the total number of reads due to the potentially extensive *thrashing* behaviour associated with backtracking, where partial solutions leading to failures are repeatedly revisited.

The predicate responsible for placing reads is shown below,

```
placeread(Seq, Part1, Part2,ReadSize,Depths) :-
    length(Seq,L),
    LMax is L - ReadSize,
    findall(X,between(0,LMax,X),AllLen),
    findall(D2,(
        between(0,LMax,X),
        length(C1,X),
        length(C2,ReadSize),
        append(C1,C2,C3),
        append(C3,_,Depths),
        D2 is min(C2))),
    MinDepths),
    inverse_depths_probs(MinDepths,Probs),
    random_select(AllLen,Probs,L1),
    L #= L1+L2,
    length(Part2,L2),
    append(Part1,Part2,Seq).
```

The `placeread/4` predicate attempts to construct a list of depths from the part of the sequence before the read, the part covering the read, and the part after the read for all combinations of possible read placements. For each of possible

read placement, it then finds the minimum depth of any position covered by the read. The call to `inverse_depth_probs/2` constructs a probability distribution over the possible read positions from the minimum depths. The probability of placing a new read of length r starting at position i, is given by,

$$P(pos = i) = \begin{cases} \frac{1}{n}, & \text{if} \sum_{i=1}^{n} d_i = 0. \\ \frac{w_i}{\sum_{h=1}^{n-r} w_h} & \text{otherwise.} \end{cases}, \text{ where } w_i = \frac{\sum_{j=1}^{n-r} \min d_j \ldots d_{j+r}}{\min d_i \ldots d_{i+r}}$$

The probabilistic selection of read positions is not implemented using the `msw` construct, but instead using the `random_select` construct, which given a list of probabilities which serves as an probability distribution over a corresponding list of outcomes, randomly selects one of the outcomes according to the probability distribution. Instead of having a fixed probability distribution, we *adapt* the probabilities in response to previous placements. This ensures that the probability of placing a new read at a given position is inversely proportional to the minimum depth of the positions covered by the read. This approach is not as declarative, since the possible outcomes and the probability distribution can only be determined during program execution. A disadvantage that comes with this lack of declarativity, i.e., that random choices occur outside the defined *PRISM model*, is that other forms of inference which rely on a fixed model are no longer possible.

The approach is very fast (almost instantaneous), but may occasionally fail. It rarely happens, but when it does, the procedure can just be restarted. It results in fewer reads and a distribution of depths/reads which appears uniformly random, except in the ends near the edges of the board which are characterized by lower depths.

4 Discussion

Our game has many similarities to Gigsaw [4], a program to generate PDF documents that include educational Next Generation Sequencing puzzles. The Gigsaw program similarly generates paper printable NGS reads, which can be aligned to or assembled into a given sequence. In differs from our program in the sense that it is simulation rather than a game. Our puzzles are more "gamified" and have multiple dependent puzzle layers, i.e., also the translation to amino acids and the quest for the secret word. Consistency between these layers is what necessitates the constraints discussed in this paper. Depth constraints are not possible with Gigsaw, which is intended to provide a realistic simulation where depth depends solely on input parameters such as the sequence length, the type and length of reads and the number of reads.

Using PRISM programs to sample biological sequence data has been done before, e.g., in [5] where they use it to generate test data to evaluate gene finders. Another application of sampling from constrained PRISM (CHRiSM) programs is the APOPCALEAPS program [6], which introduced the `soft_msw` approach.

A heuristic approach to generate programs with low probability of failure, as embodied by adaptive probability distributions is more efficient than the `soft_msw` approach in our case. However, the scheme is specific to our application and cannot easily be generalized. As a point of future research, it would be useful to develop generic, but heuristically informed methods for PLP-based random sampling which are less prone to thrashing than the pure `soft_msw` backtracking approach. Perhaps inspiration can be drawn from the methods of constraint programming [7]. Another concern that arise with sampling approaches circumventing failure, is that the probability distribution over outcomes can be skewed by the approach. This has negative consequences for using the sampling as a building block for other inference procedures. Failures can, however, be handled in the context of parameter learning by the failure adjusted maximization procedure [8] in PRISM [9] and other PLP systems [10].

References

1. Metzker, Michael L.; Sequencing technologiesthe next generation. Nature reviews genetics 11.1, 31–46. (2010).
2. Sato, Taisuke, Yoshitaka Kameya: PRISM: a language for symbolic-statistical modeling. IJCAI. Vol. 97. (1997)
3. Sato, Taisuke, et al.: PRISM Users Manual. Version 2.2. (2012).
4. Martin, D. M. A.: Gigsaw: physical simulation of next generation sequencing for education and outreach. EMBnet. journal, 18(1), 28 (2012).
5. Henning Christiansen, Christina Mackeprang Dahmcke: A Machine Learning Approach to Test Data Generation: A Case Study in Evaluation of Gene Finders. Lecture Notes in Artificial Intelligence 4571, 741–755 (2007)
6. Jon Sneyers and Danny De Schreye: APOPCALEAPS: Automatic Music Generation with CHRiSM. 22nd Benelux Conference on AI (BNAIC'10), Luxembourg (2010)
7. Dechter, Rina, and Daniel Frost: Backtracking algorithms for constraint satisfaction problemsa tutorial survey. Information-and CS Technical Report 56 (1998).
8. Cussens, James: Parameter estimation in stochastic logic programs. Machine Learning 44.3 (245-271). (2001).
9. Sato, Taisuke, Yoshitaka Kameya, and Neng-Fa Zhou: Generative Modeling with Failure in PRISM. IJCAI. (2005).
10. Chen, Jianzhong, et al. "PEPL: An implementation of FAM for SLPs. ALP Newsletter, focus on Probabilistic Prolog Systems (2011).